EE368 Project Report
Detection and Interpretation of Visual Code Markers

Taral Joglekar, Joel Darnauer and Keya Pandia
Stanford University

I. INTRODUCTION

This project aims at creating a set of image processing algorithms that can detect two-dimensional visual code markers in images and read off the bit patterns embedded in these codes. The construction of these markers is described in [1]. The images are of inferior quality, obtained from mobile phone cameras, which have high ISO noise and changing light gradient throughout the image. Also, the markers can be located at any position within the image having arbitrary scale, rotation and slight perspective distortion. This report describes the various stages of our proposed pipeline, the algorithms used, and the performance results obtained for a set of training and test images.

II. PROCESS PIPELINE

Our detection process can be broadly categorized into three main stages *viz*.
Stage 1: Image cleanup and thresholding
Stage 2: Visual Marker detection
Stage 3: Code perspective correction and readout
The following sections describe each of the three stages.

III. IMAGE CLEANUP AND THRESHOLDING

The given images are low quality with a lot of high ISO speckle noise, brightness gradients and blurring. Thus an important step before thresholding is the cleanup of these images. We tried various methods for image cleanup separately from the thresholding.

Figure 1: Original Training Images

Noise Removal: We tried removing the noise using median filtering, gaussian blurring and non-linear color aberration detection. In all cases, we realized that for most of the noise the spatial extent of the speckles was of the same order as the corner features of the visual code marker. Hence any attempt to filter out the noise selectively also adversely affected any further attempts to successful thresholding of the 'region of interest'. Hence we decided to not apply any explicit techniques to try and remove the speckle noise.

Color to grayscale conversion: The original images are color images and thus there was an initial temptation to use the extra color information available to aid in the recognition of the possible markers and non-marker false positives in the image. A major problem with this approach was that the bad chroma capture of the mobile camera meant that a color that was actually consistent was represented in the image with varying color components. For example, what should have been black color for the visual marker segments was actually captured as low intensity red or green or blue. Thus to avoid this effect, we used the value channel of the image (value from HSV decomposition) as the grayscale conversion for this image. We also tried using the average of red and green component as the gray scale value as given in [1], but found that using value worked better in some cases.

Brightness Equalization: The varying brightness can be compensated in two ways. First, we can have a specific brightness equalization stage that tries to remove the brightness gradient in the image or, secondly, we can incorporate an adaptive thresholding algorithm that accounts for the slow change in brightness within and across image scan lines. Our experiments showed that use of both these techniques gave the best results for the set of training images that we used.

Figure 2: Brightness Adjusted Images

The brightness equalization stage performs brightness equalization on parts of the image by subdividing the image into smaller blocks and scaling the block elements

so that the highest value element in any block is one. This method works very well within the block, but gives sharp edge artifacts along the boundaries of the blocks. A major problem with these edges is that they are usually more prominent than the code marker edges and create a lot of false positives. To avoid this effect we, used overlapping blocks, which reduces this effect.

Adaptive thresholding: After the brightness equalization we implemented a straight single value thresholding of the image. For this we marked anything below a brightness value below 1/2 as black, anything above as white. But this method did not work at all and we then implemented a (mean – C) algorithm. Here the threshold to use for a particular pixel was obtained by finding the mean of values in its neighborhood and subtracting a constant C from it. The problem with this approach was that fixing a single value for a single image was impossible. To compensate for this, we wrote a adaptive algorithm that varied the neighborhood size and the value of C till the number of thresholded components in the image fell below a preset value. Though this worked much better than a fixed C algorithm, we still needed to use an arbitrary value for the maximum number of components and this value seemed to be inextricably linked with the number of code markers in the image. Hence a value that worked very well with images with a single code marker worked pretty badly with images with three markers and vice-versa.

With all these failed attempts, we went back to [2] and implemented the thresholding algorithm outlined there, which basically works on a row by row basis for the image, alternating through the lines, once from left to right and then from right to left. It uses a history of (s=width/8) pixels and calculates the threshold as:

$$tval_{initial} = 0.5$$

$$tval_{new} = tval_{old} * \left(1 - \frac{1}{s}\right) + pixelval_{new}$$

Then the threshold value is used as the t^{th} percentage down value from tval. i.e.

$$threshold_{new} = tval_{new} * \left(\frac{100 - t}{100}\right)$$

The authors of [2] suggest using t = 15, and we found that the algorithm does work very well of this value. But for our images only doing this much was not enough. There were still some markers that would be too low intensity to be detected by this algorithm. As a final addition, we applied a high pass filter to the grayscale image, which compares the current pixel with a sampling of other pixels outside a 4x4 neighborhood, just to enhance the larger gradients in brightness. We use a non brightness preserving filter, so that bright areas become brighter than before and darker areas do not increase in brightness as much. This final step gave very good results for the training images.

Figure 3: Thresholded Images, notice that all the code markers are seen prominently

IV. VISUAL CODE DETECTION

Locating Candidate Guide Bars

After the black and white image has been generated, we attempt to locate any code markers that are present in the image. Before trying out new algorithms, we implemented the algorithm as suggested in [1], where we look for best fit ellipses within the regions located by the thresholding algorithm. The algorithm for detecting rectangular (or elliptical) regions of specific eccentricities relies on the first and second order moments for the regions identified. To optimize the computation of moments and to speed up the algorithm, we sift out the regions that are too small or too large to meet the criteria for the guide-bars. This would eliminate a lot surrounding regions that are a few pixels in size and also those that are too large to be considered. The moments computed from the remaining regions are primarily the mean in X and Y, the second order moment in X and Y and the covariance of X and Y. The relevant math is given in [1].

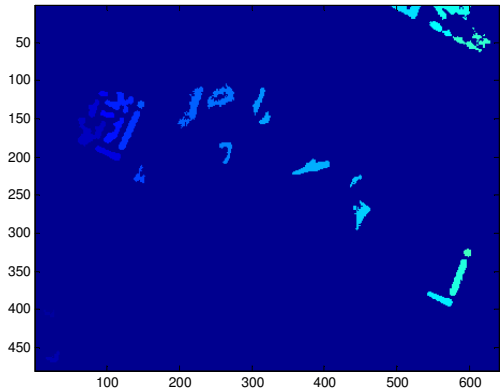


Figure 4: Regions remaining after small and large region removal

We then identify guide bars by fitting ellipses around the possible candidates and reject the candidates that do not have ellipse-eccentricities falling in the permissible range. This algorithm, however, when applied to all the training images gives a very broad range of eccentricity values [1/30, 1]. Thus a lot of unwanted regions get selected as possible guide-bar candidates. The major problem with this approach was that we could not narrow

down the eccentricity range for ellipse fitting, resulting in a lot of false positives. Hence we used a different technique for rectangle detection as outlined below.

This method begins with region detection using the *bwlabel* command and the initialization of a table containing statistics on the r regions. The erode operation is used to compute an image that is used for perimeter calculations.

The labeled image is traversed a single time pixel by pixel and the first and second order sums are accumulated into the table along with area and perimeter information and the minimum and maximum extents of the region. An earlier version of the program included a check for breaks in the scan lines but this test was removed for performance reasons.

The image scan is the most time consuming part of the program taking about 10 seconds to iterate through the pixels on the SCIEN lab machines. If Matlab implemented the table lookup as constant time operations, the algorithm should be $O(r + p)$ where p is the number of pixels in the image. To avoid the possibility that the large loop was slow because of the interpreter, we also checked a second algorithm which looped over the number of regions and used Matlab's hard coded matrix operations. This second approach had a theoretical complexity of $O(rp)$, but took only about twice as long, showing that the table lookup is a substantial part of the delay. Because of this we stuck with the initial faster algorithm.

Once the image database has been constructed we examine each region to see if it is a possible guide bar. The sums for each region are divided by the region area to generate the first and second order moments for the region. Once the second order moments are known we compute the angle of the major axis. This angle is used to rotate the moments so we can find the variance of the object along its major (v_a) and minor (v_b) axis.

Figure 5: Results of the moment-based rectangle filter. Despite the large amount of clutter in the image(gray), only the guide bars and one extraneous line are selected for further processing (black).

If the object were a perfect rectangle at some rotation, its length and width would be related to the second moments by a constant factor. We compute these putative length and width for each object at this step:

$$L = \sqrt{12 v_a + 1} \qquad (1)$$

The "+1" term is needed because the black and white image is discrete. A similar computation is performed for the minor axis (width). An advantage of this technique over the ellipse-based calculations in [1] is that it produces more accurate length estimates because it is expecting square corners. Note that this method has some vulnerability if the perspective is very steep since the codes will begin to look like parallelograms instead of rectangles. We have not observed any such problems.

After these preliminaries, we exclude objects based on a number of criteria. Objects smaller than two pixels in either dimension are excluded and so are objects near the edge of the image. The remaining objects have their measured perimeter and area compared to the area and perimeter of an ideal rectangle with the length and width computed based on moments. Objects that are not within about 50% of the predicted value are excluded. This test does a good job of eliminating objects with an Euler number other than 1 as well as objects with irregular boundaries. All of the remaining objects are marked as possible bars or cornerstones and then their aspect ratio is computed. Bars that are within about 65% of the target are marked as candidate guide bars.

Although the filter for the rectangle test has a very wide tolerance, it is surprisingly good at rejecting regions. Most images in the data set produce several hundred regions, while the number of regions that pass the rectangle test is usually less than thirty.

Cornerstone detection: Once we have a list of candidate guides we scan them to see if they fit the guide post criteria. Each end of the guide bar is checked to see if there is a candidate cornerstone. The rectangle test is applied to any object in the expected cornerstone location except that in this case the aspect ratio must be less than 1.5. If the upper-right cornerstone is detected, we look for the lower-right bar. The angle and length of this bar is used to find the position of the lower-left cornerstone.

Location of the final origin cornerstone is a little trickier. We can frame the problem as that of finding the perspective projection from 3-D to 2D of the sum of three vectors given the 2D projections of each of the vectors. Each of the three known points in the code lies on a unique ray in 3D world space. To find the position of the code marker in world space, we could reduce the problem to choosing the z-coordinate for each coordinate in world space. Since we don't care about the absolute z value only the ratios, we can fix on of the z-values at one and solve for the other two with the constraint that the two sides of the code have equal extent and are orthogonal in the 3-D world coordinate system.

In practice we found that it was not necessary to solve this system of quadratic equations and simply assume that the code markers form a parallelogram. This estimate has a larger error than other estimates, so we use a larger search radius around the estimated location and pick the

outermost cornerstone-like object in the region of interest. After this part of the algorithm is complete we have a code marker position for each corner.

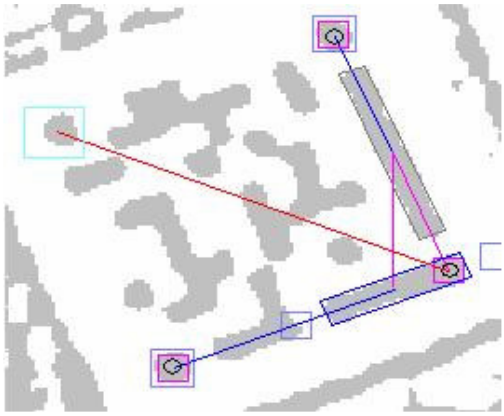


Figure 6: Results of Cornerstone finding

V. Code Bits Readout

The next step in the pipeline is to find a perspective correction (for small perspective cases only) and then transform the code from image space to the code space. We tried the algorithm given in [1] for this. We later changed this algorithm for a simpler one in which we mapped all the 121 code marker bit centers onto the original image space and readout the pixel values at those positions. Since we only had a code detection algorithm that worked for low perspective distorted images, this meant that all the bit positions were linearly spaced and we could use simple interpolation to find the bit center positions. Once the co-ordinates for the four corners have been identified, this interpolation algorithm reads out the data bits column-wise from top to bottom and then left to right from the code marker as an array of 83 consecutive bits which correspond to the data content (without the identifying guide bars bits). We place these bits into the appropriate positions in the 11x11 grid to obtain the complete marker array or image. As mentioned, this fails for extreme perspectives. The result of this algorithm on the code marker shown in Figure 6 is shown in Figure 7.

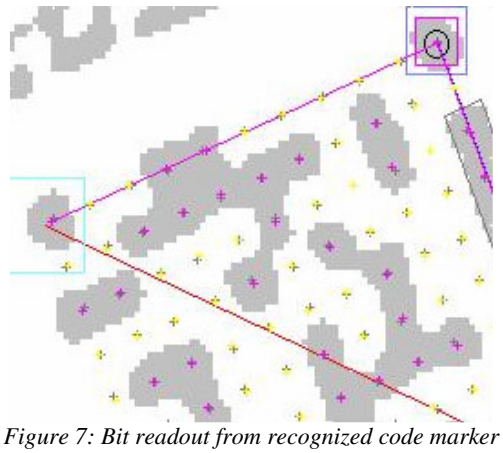
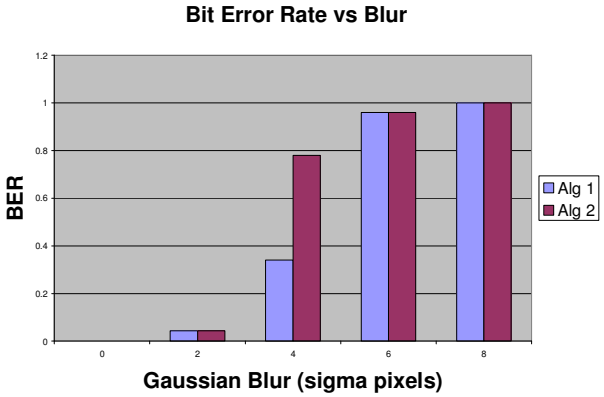


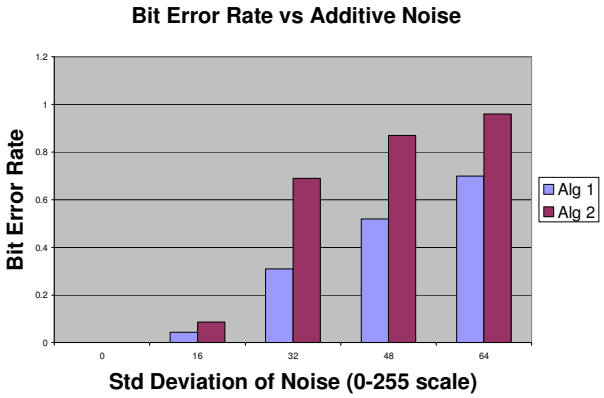
Figure 7: Bit readout from recognized code marker

VI. Performance Metrics

After some bug fixes we were able to get our image pipeline working with zero errors on the training images. We then proceeded to generate some test images that would stress other dimensions such as extreme perspective and strange illumination conditions. These test cases revealed some additional bugs and some limitations with the initial thresholding algorithms especially on very bright images. We decided that we needed a way to compare algorithms that would get a perfect score on the test images in some standardized way. To accomplish this we wrapped the evaluation script in a routine that degrades the test image with a variable amount of gaussian blur and additive noise. By sweeping these parameters we can get a bit-error-rate versus blur/noise characteristic for a given algorithm over the whole training data set.

The following figure shows the bit error performance for the original data set for two candidate image thresholding algorithms.
Alg 1: The final algorithm that we used.
Alg 2: This algorithm uses soft-coring to reduce noise and then selects a threshold by finding a weighted mean in the neighborhood of each pixel. The weighting function is the Hann window.





Both the above algorithms achieved perfect scores in the test images, but Algorithm 2 is more susceptible to noise and blurring effects than Algorithm 1.

Table below gives performance metrics – Error statistics, Euclidean distance and run-time for each one of the given images in the training set provided to us. There we no false positives or negatives.

Image Number	Time taken	Avg. Error in dist of Origin location	No of correct bits	No of bits detected
1	16	0.8	83	83
2	14	1.1	166	166
3	12	1.2	249	249
4	15	1.1	83	83
5	18	1.1	249	249
6	12	1.4	83	83
7	16	1.2	166	166
8	15	0.8	83	83
9	15	1.1	249	249
10	21	0.6	249	249
11	13	1.8	83	83
12	19	0.2	166	166

Histogram Considerations for Adaptive Thresholding and Contrast and Brightness Adjustment: As part of our test routine, we created some test images from our camera phone and tried running our algorithm on those images. We observed cases where the algorithm failed and tried to look into possible reasons and fixes for the problems that we encountered. The primary cause for concern was that the images obtained from our camera were far brighter were basically histogram equalized as compared to the images in the training set provided to us. Histograms of two sample images (one from the training set and one from our set of images) are given below. It is very evident that they are fundamentally different in their nature.

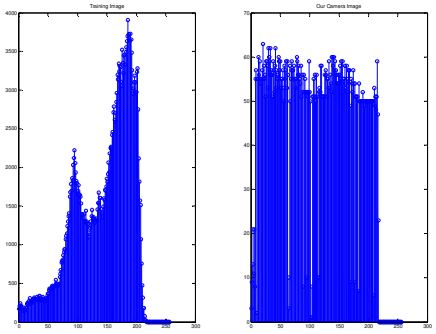


Figure 8: Histogram comparison of training and test images

We tried some brightness reduction and contrast stretch on our images and observed a marginal improvement in performance.

Breaking cases for our algorithm: From the test images that we obtained from our camera phone, we concluded that our algorithm fails for the following extreme case:

- *Extreme Perspective:* Since our algorithm works on a parallelogram approximation of the code marker, an extreme perspective breaks the algorithm.
- *Extreme overall image brightness:* This causes our adaptive threshold algorithm to break because there no longer is sharp enough contrast for the marker read-out and segmentation.
- *Marker Identifiers less than 2 pixels wide:* Trying to detect markers below this threshold is susceptible to noise and hence false positives.
- *Markers within 10 pixels of the image boundary:* We added this condition to remove edge artifacts of brightness enhancement near the image boundaries.

VII. CONCLUSION

The algorithm designed gives high accuracy and excellent performance for images with suitable resolution, clarity and contrast to give an efficient identification and read-out of marker bits. The algorithm may be modified to work on images fundamentally different from the ones it is trained to decipher. Further improvement could incorporate intelligent routines that automatically adjust to varying resolutions, contrasts and image qualities. This could be very useful as different cameras have very diverse ranges of image quality and resolution.

REFERENCES

[1] Michael Rohs, "Real-World Interaction with Camera Phones", Institute of Pervasive Computing, Swiss Federal Institute of Technology, Zurich.

[2] Pierre D.Wellner: "Adaptive Thresholding for the DigitalDesk", Technical Report EPC-93-110, Rank Xerox Research Centre, Cambridge, UK, 1993.

[3] Michael Rohs, "Marker Based Interaction Techniques for Camera Phones", http://www.inf.ethz.ch/personal/rohs/visualcodes/rohs-mu3i-final.pdf

[4] Jun Rekimoto et al., "CyberCode: Designing augmented Reality Environments with Visual Tags", http://www.csl.sony.co.jp/person/rekimoto/papers/dare2000.pdf

APPENDIX 1

Our Test Images

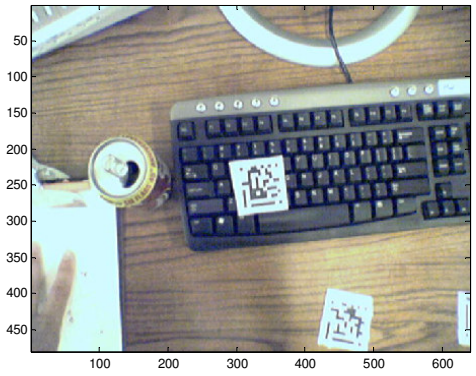
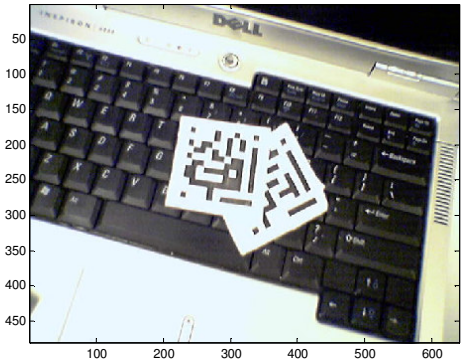


Image with high brightness



Algorithm did not give a false positive for the partially hidden code marker

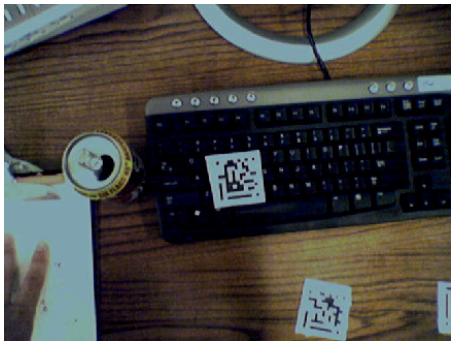
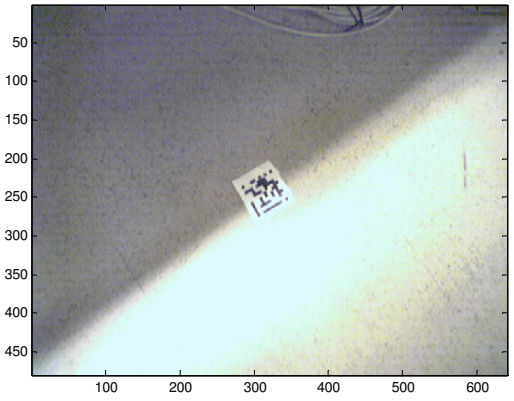
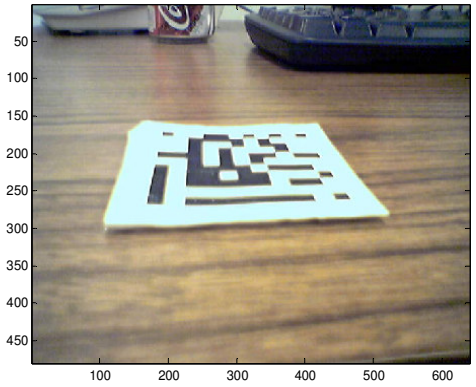


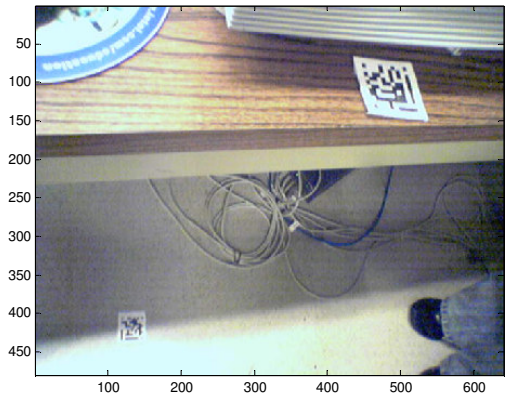
Image brightness adjusted to match training image brightness



Algorithm worked in spite of large brightness variance



Algorithm fails for such high perspectives



Algorithm worked for the larger marker but not for the smaller one

APPENDIX 2: Log of Work Done

Taral Joglekar

- Tested various methods for noise removal, but these were not included in the final submission
- Wrote the brightness adjustment algorithms for overlapped and non-overlapped blocks
- Image prefiltering for adaptive thresholding by using blurring filter for removing hard edges followed by a variation of the edge filtering to enhance larger brightness changes
- Adaptive Thresholding using algorithm given in [1]

Joel Darnauer

- Wrote the guidebar/cornerstone finder including the computation of moments and the rectangle filter
- Wrote the graphics debug routines
- Wrote an alternative thresholding algorithm based on soft-coring and a 2D thresholding filter. This algorithm was "Alg 2" from the paper and was not included in the final submission.
- Wrote the algorithm comparison program which feeds noisy/blurry images and measures bit error rate.

Keya Pandia

- Wrote a preliminary Adaptive Thresholding algorithm to support the algorithm to convert the RGB color image to a good contrast black and white image
- Designed an algorithm for region elimination based on sizes
- Designed the tables for moments and code to compute covariance and centroids and the code for region selection based on ellipse fitting and selecting the range of eccentricities (which wasn't eventually included in the final project)
- Wrote the code for interpolating the positions of pixels for bit-readout and the algorithm to read out the information bits
- Worked on image contrast and brightness adjustment to investigate the efficiency of the code for varying image brightness and contrast.